TOWARDS OBJECT ORIENTED CONCURRENT PROGRAMMING(Software Science and Engineering)

AUTHOR(S):

Yonezawa, Akinori; Matsuda, Hiroyuki

数理解析研究所講究録
547巻 1985年 1-22

TOWARDS OBJECT ORIENTED CONCURRENT PROGRAMMING

Akinori Yonezawa and Hiroyuki Matsuda

Department of Information Science
Tokyo Institute of Technology
Ookayama Meguro-ku, Tokyo 152

Contents

1. Introduction

Objects in the object oriented programming are conceptual entities which model the functions and knowledge of "things" that appear in problem domains. The fundamental aim in the object oriented programming is to make the structure of a solution as natural as possible by representing it as interactions of objects.

Currently proposed formalisms for object oriented programming confine themselves in the sequential world. This is too restrictive. Parallelism is ubiquitous in our problem domains. Behaviors of computer systems, human information processing systems, corporative organizations, scientific societies etc. are results of highly concurrent (independent, cooperative or contentious) activities of their components. To model and study such systems, or to solve problems or design systems by the metaphore of such systems, it is necessary to create adequate formalisms in which various concurrent activities and interactions of "objects" can be naturally expressed and which are executable as computer programs.

Our present work is to propose such a formalism. The problem domains to which we intend to apply our formalism include problem solving and planning in AI, building expert systems, modeling human cognitive processes, designing real-time systems and operating systems, and designing and constructing office information systems.

2. A Model of Computation

The framework of our present work is based on the Actor computation model proposed by C. Hewitt [HE73][HE77] in early 70's. In this model, computations are performed by concurrent message passing among procedural objects called actors. Actors model conceptual or physical entities which appear in problem domains. Messages specify requests, inquiries or replies.

Each actor has its own processing power and it may have its local memory. An actor is always in one of two modes, active or inactive and it becomes active when it receives a message. Each actor has its own description which determines what messages it can accept and what computations it performs. Upon receiving a message, an actor can make simple decisions, send messages to actors (including itself), create new actors and change its local memory according to its description. After performing the described computation, an actor becomes inactive until it receives the next message.

Though message passings in a system of actors may take place concurrently, we assume message arrivals at an actor be linearly ordered. No two messages cannot arrive at the same actor simultaneously and a single message queue sorted in the arrival order is assumed for each actor. When a message arrives at an actor, if the actor is not active and no messages are in the queue, then the message is received by the actor. If the actor is active or messages are in the queue, the message is put at the end of the queue.

There are two classes of actors, "serialized" actors and "unserialized" actors. A serialized actor is activated by one message at a time. While a serialized actor is being activated by a message, it is locked and cannot receive the next message. We do not assume this property for unserialized actors. In the subsequent discussion we focus our attention on serialized actors.

Each actor has its own local clock (frame of time reference) which is in accordance to the advancement of its local computation. In general, local clocks associated with different actors may not agree. No uniform global clock is assumed.

The above account of the actor computation model is a very intuitive one. For those who are interested in its mathematical foundation, please look into the two reports [CL81] and [AG84].

3. Types of Message Passings and Continuations

To study the versatility of the actor model of computation, we modeled and described various parallel or real-time systems using a simple set of notations. In the course of this process, we made a simple assumption on the message arrival and found it sometimes necessary to distinguish three types of message passings which are not included in the original Actor model of computation.

[Assumption]

> When two messages are sent to an actor T by the same actor A, the time ordering of the two message transmissions (according to A's clock) must be preserved in the time ordering of the two message arrivals (according to T's clock).

This assumption was not included in the original actor model of computation[HB77](1), but without this, we cannot, for example, model a computer terminal or displaying device as an actor which receives lines of characters as messages that are sent by an actor modeling an output handling program in an operating system.

["Past" Type Message Passings]

> Suppose an actor A is being activated and it sends a message M to an actor T. Then A does not wait for M to be received by T. It just continues its computation after the transmission of M (provided that the transmission of M is not the last action during the current activation of A).

---

(1) In computer networks of large scale such as ARPA net, this assumption is not necessarily guaranteed due to complex routing algorithms which take into account of node failure and load balance.

We call this type of message passings "past" type because sending a message finishes before it causes the intented effects to the message receiving actor. Let us denote a past type message passing by the following notation.

[T <= M] (1)

The past type corresponds to the situation where one requests or commands someone to do some task and simultaneously he proceeds his own task without waiting for the requested task to be completed. This type of message passings substantially increase concurrency of activities within a system. (Past type message passings can further be divided into two kinds which may reflect two different implementation strategies. In one kind, an actor which transmits a message does not continue its computation until the arrival of the message is assured, while in the other kind, the actor continues its computation as soon as the message leaves the actor. Of course the latter kind allows higher concurrency than the former one, but may sacrifice the robustness against various unexpected errors in the system's components.)

["Now" Type Message Passings]

When an actor A sends a message M to an actor T, A waits for not only M to be received by T, but also waits for T to return some information to A. If T does not return anything, A waits until T's current activation caused by M ends.

This is similar to ordinary function/procedure calls, but it differs in that T's activation does not have to end with sending some information back to A. T continues its computation during the same activation caused by receiving M. A now type message passing is denoted by

[T <== M] (2)

Returning information from T to A may serve as an acknowledgement of receiving the message (or request) as well as reporting the result of a requested task. Thus the message sending actor A is able to know that his message was surely received by the actor though he may waste time in waiting. The returned information (certain values or signals) is denoted by the same notation as the message passing. Namely, (2) denotes not merely an action of sending M to T by a now type message passing, but also denotes the information returned by T. If the activation of T ends without returning any information, we assume, by convention, (2) denotes some special value (e.g. nil).

Now type message passings provide a quite convenient means to synchronize concurrent activities performed by independent actors when it is used together with the parallel construct that will be discussed in a later section. (It should be warned that recursive now type message passings cause local deadlock.)

["Future" Type Message Passings]

Suppose an actor A sends a message M to an actor T expecting a certain requested result to be returned from T. But A does not need the result

immediately. In this situation, A does not have to wait for T to return the result after the transmission of M. It continues its computation immediately. Later on when A needs that result, it checks A's internal memory area that was specified at the time of the transmission of M. If the result is ready, it is used. Otherwise A waits there until the result is obtained

A future type message passing is denoted by

[x := [T <= M]] (3)

where x is the specified memory area (or a variable). A system's concurrency is increased by the use of future type message passings. If the now type were used instead of future type, A has to waste time by waiting for the currently unnecessary result to be produced. Future time message passings have been incorporated in previous actor based programming languages [LI81][FU84].

In the above discussion, the contents of a message was left vague. We should make it clear, in order to make a more precise account of how various information flows among actors through message passings. A message consists of two parts, an RR-part and a C-part. An RR-part which stands for a request/reply part tells the message receiving actor about the contents of a request or it is used to carry a reply, answer or result of a requested task.

When a message is sent by a "past" type message passing to request an actor to do some task, it is sometimes useful for the message sending actor to have a means to specify a destination actor where the result of the requested task should be sent. We call this destination actor a "continuation". A C-part which stands for a continuation-part provides this means. (Without an explicit indication of the destination, only thing one can do is either to have the actor which carries out the requested task keep the result within in itself, or to have the result sent to some default actor.) In our notational convention, a message is expressed by a pair whose first and second parts are separated by a period. The first part and second part correspond to its RR-part and C-part, respectively. Namely, it is of the form

[<RR-part> . <C-part>].

When the C-part of a message need not be specified, it is left blanked. In this case the message is a Lisp singleton list of the following form

[<RR-part>]

where the period after <RR-part> is omitted. In fact, the C-parts of messages sent by "now" or "future" types must be void, because the destination to which the result is supposed to be sent is predetermined. Namely, a "future" type message passing itself specifies an internal memory (or a variable) of the message sending actor. For the case of "now" type, we can view this type of message passings as a special case of "past" type message passings with a certain continuation attached. Since we need some more notational conventions (or rather a language) to describe the definition of this continuation, the actual definition is given in the Appendix. (The following section is a prerequisite

for the Appendix.)

4. Defining and Creating of Objects

In order to describe behaviors of actors in more precise and concrete terms, we need to develop a language. We have tentatively designed and implemented a programming language called ABCL (Active object Based Core Language). The purpose of designing this language is manifold. It is intended to serve as an experimental programming language to construct software in the framework of object-based concurrent programming. The kind of the application domain we emphasize includes the AI fields and we plan to use this language as an executable thought-tool for developing the paradigm of distributed problems solving[SI81] and cognitive models. It is also intended to serve as an executable language for modeling and designing of various parallel or real time systems. Direct derivatives of this language have been used to write natural language parsing/understanding systems[OY84][MI85] as well as modeling various parallel or real-time systems, some of which will be mentioned in this paper.

The primary design principles of this language are summerized by the following two points.

[1] Clear semantics: the semantics of the language should be as close to the simple underlying computation model as possible.

[2] Practicality: for AI programming, various features of Lisp can be utilized to exploit efficiency and programming ease as long as the framework of the object oriented programming is maintained.

The purpose of the present paper is not to introduce the details of the language, we keep its explanation minimum. For those who are interested in the language, see the companion paper[YM84].

4.1. Defining Objects

Each actor has a fixed set of message patterns it accepts. To define the behavior of an actor, we must specify what computations or actions it will perform for each such message pattern. The description of computation for each message pattern is called a "script". If an actor has its local memory, its computations may be affected by the current contents of such memory. Thus in order to define an actor with local memory, we must also describe how the actor's local memory is represented. Representations of local memory are variables or internal actors which have its own local memory.

To define a (serialized) actor, ABCL uses a notation of the following form. (state: ...) declares the representation of local memory and initializes it.

```
[object <object-name>
    (state: ...
        <representation of memory>
            ...                    )

   (=> [<pattern>]   <script>   )                                    (4)
    ...
    ...
   (=> [<pattern>]   <script>   )    ]
```

Scripts are basically written in terms of message passings of the three types, referencing to variables and calculating values or manipulating list structures using Lisp functions. These actions are performed sequentially unless special parallel execution constructs are used.

As an illustrative example, let us consider an actor which models the behavior of a semaphore. A semaphore has a counter to store an integer with a certain initial value (say 1) and also it has a queue for waiting processes which is initially empty. We represent the counter as a variable and the queue as an (internal) actor which behaves as queue. A semaphore accepts two patterns of messages, [p-op: . C] and [v-op: . C] which corresponds to the P-operation and V-operation. In ABCL, symbols ending with a colon in messages or message patterns are constants, whereas symbols starting with a capital letter are pattern variable which bind components of incoming messages. (p-op: and v-op: are constant. C is a pattern variable which binds the C-part, namely the continuation of an incoming matching message.)

Using the notation (4), a definition of the semaphore actor is shown below.

```
[object aSemaphore
    (state: [counter := 1]                          ; := means assignment.
            [process_q = [CreateQ <== [new:]]])     ; = means binding.

  (=> [p-op: . C]  <script for P-operation>     )

  (=> [v-op: . C]  <script for V-operation>     ) ]
```

Note that a "now" type message passing is used to create a queue actor and it is bound to a symbol process_q.

4.2. Creating Actors

CreateQ in the above example is an actor which creates and returns a new actor which behaves as a queue. We assume it is defined elsewhere. CreateQ can be viewed as a class of queues and the created queue actor as an instance of the queue class if we use the terminology of AI or SmallTalk[GB83]. In ABCL, rather complicated notions such as classes and meta-classes are unified as the notion of actors, which allows us to manipulate classes and meta-classes as objects.

Actors which create and return an actor are often defined in the following fashion.

```
[object CreateSomething
  (=> [<initial-information> . <continuation>]

    [<continuation> <= [[object ...          ; a newly created actor is
                          (=> [...] ...)     ; sent to <continuation>
                           ...
                          (=> [...] ...)]
                       .
                       nil]]
  )]
```

Namely, a message whose RR-part is a newly created actor defined by [object ....] and whose C-part is nil is sent to <continuation>. Creating a new actor and sending it back to the continuation is one of typical situations where a message with its C-part being nil is sent to the original continuation. (By the original continuation, we mean the continuation (namely the C-part of a message) that causes this typical message transmission.) A simple abbreviated notation in ABCL expresses this scheme of message transmission.

```
(=> [<request>]     ... !<expression> ...),
```

which is equivalent to

```
(=> [<request> . <continuation>]
       ...  [<continuation> <= [<expression> . nil]] ...).
```

5. Parallelism and Synchronization

5.1 Parallelism

Using the abbreviated notation explained in the previous section, the actor which creates and returns a semaphore actor is defined in Figure 1.

```
[object CreateSemaphore
  (=> [init: N]

    ![object                        ; definition of a semaphore actor begins.
        (state:
          [counter := N]
          [process_q = [CreateQ <== [new:]]])

       (=> [p_op: . C]
         [counter := (sub1 counter)]
         (case (> 0 counter)
            (is t              ; if counter is negative
              [process_q <= [enqueue: C])
            (otherwise
              [C <= [go:]])))

       (=> [v_op: . C]
         [counter := (add1 counter)]
         (case [process_q <== [dequeue:]]
          (is nil                          ; if process_q is empty
              [C <= [go:]])
          (is FrontProcess     ; the head of process_q is bound to FrontProcess
            { [FrontProcess <= [go:]],
              [C <= [go:]] }   )))
     ]  )]
```

Figure 1. Defining a Semaphore Actor

In the script for v-op:, (add1 counter) is an invocation of a lisp function add1 and the result updates the contents of counter. When process_q actor is empty, a [go:] message is sent to the continuation which is bound to C; otherwise the first process that has been waiting is removed from the queue and [go:] messages are sent to this process and the continuation simultaneously. (Notice that processes are modeled as actors.) As noted earlier, a script is usually executed sequentially. But when a special construct denoted by

```
{ E1 , ... , Ek }
```

is executed, the executions of E1,...,Ek start simultaneously. The execution of this construct, which we call a parallel construct, does not end until the executions of all the components E1,...,Ek do not end. When the components of a parallel construct are all past type message passings, the degree of parallelism caused among the message receiving actors is not much greater than the degree of parallelism caused by the sequential execution of the components because time cost of a message transmission is very small. But if a parallel construct contains now type message passings, the possibility of exploitation of parallelism among the message receiving actors is very high.

After having explained parallel constructs, it is an appropriate time to review the basic types of parallelism provided by ABCL.

[1] concurrent activations of independent actors.

[2] parallelism caused by past type message passings.

[3] parallelism caused by parallel constructs.

5.2. Synchroniztion

Parallel constructs are also powerful in synchronizing the behaviors of actors because the semantics of a parallel construct requires that its execution completes only when the executions of all the components complete. When a parallel construct contains a now type message passing in a script, all the intended activations of the message receiving actors must be completed before going on to the execution of the rest of the script after the parallel construct. (Note that we need no synchronization if all the components of a parallel construct are past type message passings.)

For example, suppose the movements of a robot arm are actuated by three step motors, each being responsible for the movement along different coordinates[KS84] and for each motor there is an actor operating it. In order to pick up something by the fingers attached to the arm, the control program sends signals to the three actors in parallel, and it must wait until the rotations of all the three motors stop. See a fragment of the program below.

```
...

{ [motorX <== [step: 100]],
  [motorY <== [step: 150]],
  [motorZ <== [step: -30]] }
<command to pick up>
...
```

We conclude this section to remind one that ABCL provides the following four basic mechanisms for synchronization.

[1] serialized actor: the activation of a serialized actor takes place one at a time and a single first-come-first-served message queue is associated with each actor.

[2] now type message passing: it does not end until a certain result is returned or the activation of the receiving actor comes to end.

[3] future type message passing: when the specified variable is referred to, the execution is suspended if the contents is not updated yet.

[4] parallel construct: as discussed above.

Although we have shown an implementation (or modeling) of semaphores in terms of the actor paradigm, we think semaphores are too primitive and unstructured as a basic synchronization mechanism. Thus we have no intention of using semaphore actors to synchronize behaviors of actors. Our experience of writing programs which require various types of synchronization suggests that combinations of the four mechanisms listed above seem sufficiently powerful for dealing our current problems.

6. Distributed Problem Solving

In this section, we present a simple model of distributed problems solving described in ABCL in order to show the simplicity and naturalness of our object oriented concurrent programming paradigm.

6.1. A Problem Solving Organization

Suppose a high level manager is requested to create a project team to solve a certain problem within a certain time limit. He first creates a project team of k problem solvers with different problem solving strategies and then he creates a project manager who dispatches the same problem to each problems solvers. For the sake of simplicity, they are assumed to work independently in parallel. If one of them has solved the problem, it must report the solution to the project manager immediately. When the project manager receives a success report, it then sends a "kill" or "stop" message to all the problem solvers. If nobody can solve the problem within the time limit, the project manager sends a failure report to the high level manager and commits suicide. This problem solving organization is easily modeled and expressed by ABCL without any structural distorsions. One is invited to describe this problem solving organization using other formalisms mentioned in 8.2.

We first define an actor which create a manager for parallel problems solvers in Figure 2. This manager corresponds to the project manager. The initial information for this actor includes a problem description, a group of problem solvers (namely, the project team), a time limit and the name of a person to whom the result should be reported (namely, the high level manager).

```
[object Create_Manager_for_Parallel_Problem_Solvers
  (=> [problem_description: PD
       solvers: SVS
       time_limit: N
       reported_to: C]

    ![object Me              ; this actor acts as the project manager.
        (state:
           [time_keeper = [Create_Alarm_Clock <== [wake: Me after: N]]])

       (=> [solve:]
         [SVS <= [broadcast: [solve: PD] . Me]]
         [time_keeper <= [start:]])

       (=> [wake:]       ;time is up before somebody solves the problem in time
         [SVS <= [broadcast: stop:]]   ; all the solvers have to stop
         [C <= [nobody_solved:]]  ; report failure to the high level manager
         (suicide))            ; and commits suicide.

       (=> [i_solved: solution: S]   ; a solver claimed to solve the problem
         [SVS <= [broadcast: stop:]]
         [C <= [solved: solution: S]]) ]      )]
```

Figure 2.

6.2. Alarm Clocks

When a manager actor (the project manager) is created, it creates an alarm clock called time_keeper who knows whom and when to wake. When the manager actor receives a message [solve:] from the high level manager, it sends to the project team actor a message that contains the problem description and sets the alarm clock. After doing these two actions, it sleeps, namely becomes inactive.

The project team is represented as a broadcaster actor which contains a collection of actors and sends (broadcasts) a received message of the form [broadcast: ...] to each member of the collection. See Figure 3. By sending [add: ...] and [delete:...] messages, we can add and delete members. Note that a message sent to the project team may contain an explicit C-part (continuation). This will specify to whom each problem solver sends a report.

```
[object Create_Broadcaster
  (=> [new:]

    ![object                        ; definition of a broadcaster actor begins.
         (state:
            [receivers := nil]) ; a collection of actors to get broadcast.
                                ; it is initially nil
       (=> [add: A_receiver]
         [receivers := (cons A_receiver receivers)])

       (=> [delete: A_receiver]
         [receivers := (delete A_receiver receivers)])

       (=> [broadcast: Message . C]
         (do ((l receivers (cdr l)))
             ((null l))
              [(car l) <= [Message . C]]))))] )]
```

Figure 3

A definition of an actor creating alarm clock actors is given in Figure 4. Notice that an alarm clock actor sends a [start:] message to itself after a unit of time is spent and repeats this cycle until a specified number of time units elapses.

```
[object Create_Alarm_Clock          ; to create an actor which
  (=> [wake: Person after: N]      ; wakes up a person after N time units.

    ![object Me                     ; definition of an alarm clock actors begins
         (state: [count := 0])
       (=> [start:]
         (consume_a_unit_time)     ; invoke some procedure to spend a unit time
         [count := (add1 count)]
         (case (> count N)
           (is t
             [Person <= [wake:]] )
           (otherwise
             [Me <= [start:]])) )]   )]
```

Figure 4.

When the project manager receives a [wake:] message from the alarm clock, it sends a [stop:] message to the project team to stop all the problem solvers because receiving the [wake:] message means nobody could solve the problem within the time limit. Then it sends a failure report to the high level manager and kills itself.

When the project manager receives an [i-solved:...] message, this means that one of the problem solvers have succeeded in solving the problem. Thus it sends a [stop:] message to the project team and make a success report to the high level manager.

In Figure 5, we give the definition of a typical problem solver actor. Its local memory is used for a flag to indicate the completion of problem solving and it is also used for recording the state of progress of problem solving. When it receives a [solve: ...] message, it tries to solve a specified problem. When it has solved the problem, it sends the description of the solution to the continuation that is bound to To_be_reported, which should be the project manager. If the problem is not solved yet, it records the current state of progress and sends a [go_on:] message to itself.

```
[object A_Problem_Solver
    (state: [solved := "not_yet"]
            [state_of_progress := nil])

  (=> [solve: Problem_Description . To_be_reported]

    ...  trying to solve the problem ...

    (case solved
      (is t
        [To_be_reported <= [i_solved: solution: <description of solution>]])
      (otherwise
        [state_of_progress := <information for restarting>]
        [A_Problem_Solver <= [go_on:]])) )

  (=> [go_on:]

     ... restart the problem solving using "state_of_progress"
         and then review the progress . ..
    (case solved
      (is t
        [To_be_reported <= [i_solved: solution: <description of solution>]])
      (otherwise
        [state_of_progress := <information of restarting>]
        [A_Problem_Solver <= [go_on:]])) )

  (=> [stop:]       ; [stop:] messages are sent from
    (suicide)) ]    ; the broadcaster actor (i.e. the project team)
```

Figure 5

6.3 .Interruption

It is important to send a [go_on:] message to the problem solver actor itself by a past type message passing. This means that the problems solver temporarily stops his work. While the problem solver is busy working, the project manager

might have sent a [stop:] message to it via the broadcaster actor and the [stop:] message might have been put in the arrival message queue of the problem solver. In such a case, the [stop:] message might be put before the [go-on:] message in the arrival message queue, which implies that the [stop:] message is received by the problems solver before the [go-on:] message.

In general, when an actor is continuously working or active, there is no chance of interrupting it by sending messages. If one wishes to interrupt an activation of an actor, the structure of its behavior must be designed in such a way that the actor should review itself the progress of its task after a certain amount of effort is spent. And if the progress is not sufficient, the actor sends a restart message to itself by a past type message passing and becomes inactive. By doing so, we give an interrupt message a chance of being received by the actor.

The above approach is that the programmer is responsible for making actors interruptable when designing them. Another approach might be to let a special message have a privilege to interrupt activations of actors. We did not take the latter approach because it would obscure the semantics of the language considerably.

Returning to the explanation of our modeling of the original problem, we give the definition of the high level manager actor in Figure 6, which should be self-explanatory.

```
[object Higher_Manager
    (state:
      [project_team = [Create_Broadcaster <== [new:]]])

  (=> [start_a_new_project: Problem time_limit: N]
    (temporary: a_manager)
        ... creating problem solvers ...
    [project_team <= [add: problem_solver1]]
      ...
      ...
    [project_team <= [add: problem_solverk]]
    [a_manager = [Create_Manager_for_Parallel_Problem_Solvers
                        <== [problem_description: Problem
                             solvers: project_team
                             time_limit: N
                             reported_to: Higher_Manager]]])

    [a_manager <= [solve:]]))

  (=> [solved: solution: S]     ...       )    ;receiving a success report

  (=> [nobody_solved:]        ...       )]     ;receiving a failure report
```

Figure 6

7. Inheritance Mechanisms

Unlike other object oriented languages, the current version of ABCL does not have language-predefined mechanisms for "inheritance". It is easy to incorporate the simple inheritance mechanism of SmallTalk[GR83] into our language. But we are not convinced whether the single inheritance mechanism is sufficient enough. At the same time we feel that various features provided by the multi-inheritance mechanisms of Flavors[WD81] are rather complicated and difficult to manage. Since ABCL can not only easily simulate currently proposed various inheritance mechanisms, but also it can create instances of the same class which have different sets of so called "super classes", we are not urgent to provide built-in mechanisms of inheritance in our language.

In Figure 7, we give an ABCL definition of a class A whose instances may have different super classes. (The semantics of ABCL requires that an incoming message is matched against the patterns sequentially top to bottom. Thus [Any . C] can be matched against a message that do not match any previous patterns.) The reason why this is possible is that "instances", "classes", and "meta-classes" are uniformly viewed as actors and thus they can be operated in the same manner as "objects".

```
[object CreateA
  (=> [new: MySuperClass]          ; super class is parametrized.
    (temporary:
      [MySuperClassInstance = [MySuperClass <== [new:]]])

    ![object          ; This actor corresponds to an instance of the class A.
        ...
      (=> [...]              ...              )

      ...

      (=> [Any . C]
        [MySuperClassInstance <= [Any . C]])]  )]
```

Figure 7

8. Concluding Remarks

In place of conclusions, we will comment on important issues that we have left undiscussed.

8.1. Programming Environments and Implementation

The first stage of (concurrent) programming in the object oriented style is to determine, at a certain level of abstraction, what kind of objects are necessary and natural to have in solving the problem concerned. At this stage, message passing relations (namely what objects send messages to what objects) are also

determined.

Since it is often useful or even necessary to effectively overview the structure of a solution or result of modeling, those identified objects and message passing relations should be recored and be retrieved or even manipulated graphically. For this purpose, we are currently designing and implementing a programming aid system on a SUN-II Workstation with a multi-window system and a standard pointing device[KA85]. A typical action using this system might be to add a node to a graph representing message passing relations among objects (where nodes correspond to objects), point the node by a mouse to get a pop-up menu and select/perform operations such as editing and compiling for the code of object.

In general, debugging concurrent programs is a rather difficult task. One example of debugging aids we are using is a local history option. When this option is set, for each specified actor, a chronological history of incoming and outgoing messages with states of its local memory recorded. In our implementation, the local history of each actor is stored in the local memory of each actor and it can be retrieved and tailored by editing operations to provide various debugging information.

It should be noted that serial versions of ABCL are implemented with a handy compiler-compiler system developed by the second author[MA84].

8.2. Comparison to Other Work

Our present work is related to a number of previous research activities. To distinguish our work from them, we will make brief comments on the representative works.

CSP[HO78] :

Dynamic creation of processes is not allowed. Message passing relations among processes must be predetermined and cannot be changed. Sending and receiving must be synchronized (handshake). All these restrictions are not imposed on ABCL.

Monitors[HO74] :

The property that a monitor procedure can be executed by only one process at a time is similar to that of serialized actors. But the basic mode of communication in programming with monitors is the call/return bilateral communication. Past type message passings accompanied with continuations in ABCL give us more flexibility and expressive power. See [YO79] for more complete discussion on this subject.

Concurrent Prolog[ST83] :

Channel variables must be explicitly merged for an object to receive messages from more than one object.

SmallTalk80[GB83], LOOPS[BS81] :

Very limited primitives for concurrency control are provided. Classes and meta-classes cannot be treated as objects (i.e. they cannot be sent in messages).

8.3. Other Examples

A wide variety of small example programs have been written in ABCL and we are fairly convinced that essential part of ABCL is robust enough to be used in the intended areas. Examples we have written include parallel discrete simulation[YO84a], deamons, blackboard models, production rules, production systems, robot arm control, mill speed control[MA94], bounded buffers, integer tables[HO74], simulation of data flow computations, process schedulers etc. Some of these example programs will be found in [YM84].

Acknowledgements

Almost daily discussions with S. Fukui, K. Mitsui, Y. Kuno, J. Rekimoto, S. Kanada, and I. Ohsawa were stimulating and valuable in fermenting various ideas. We deeply appreciate them.

References

[AG84] Agha, G.: Semantic Considerations in the Actor Paradigm of Concurrent Computation, Draft May (1984).

[BS81] Bobrow, D. and Stefik, M: The LOOPS Manual, Memo KB-VLSI-81-13, Xerox PARC, (1981).

[BG84] Broda, K. and Gregory, S.: Parlog for Discrete Event Simulation, Proc. 2nd Int. Logic Programming Conf., (1984).

[CL81] Clinger, W.: Foundations of Actor Semantics, Ph.D. Thesis, MIT, (1981), available as TR-6333 MIT AI Lab.

[FU84] Fukui, S.: An Object Oriented Parallel Language, Proc. Hakone Programming Symposium, (1984), in Japanese.

[GR83] Goldberg, A. and Robson, D.: SmallTalk80 - The Language and its Implementation - -, Addison Wesley, 1983.

[HB77] Hewitt, C. and Baker, H.: Laws for Parallel Communicating Processes, IFIP-77, Toronto, (1977).

[HE73] Hewitt, C. et al.: A Universal Modular Actor Formalism for Artificial Intelligence, Proc. Int. Jnt. Conf. on Art. Int., (1973).

[HE77] Hewitt, C.: Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence, Vol. 8, (1977), pp.323-364.

[HO74] Hoare, C.A.R.: Monitors: An Operating System Structuring Concept, CACM, Vol. 17, No. 10, (1974), pp.549-557.

[HO78] Hoare, C.A.R.: Communicating Sequential Processes, CACM. Vol. 21, No. 8., (1978), pp.666-677.

[KS84] Kerridge, J. M. and Simpson, D.: Three Solutions for a Robot Arm Controller Using Pascal-Plus, Occam and Edison, Software - Practice and Experience - Vol. 14, (1984), pp.3-15.

[KA85] Kanada, S: Master Thesis, Dept . of Information Science, Tokyo Institute of Technology, in preparation.

[LI81] Lieberman, H.: A Preview of Act-1, AI-Memo 625, MIT AI Lab., (1981).

[MA84] Matsuda, H.: A Language Description Language LEAG and its Applications, Preprint of WGSF, IPSJ, September, (1984).

[MI85] Mitsui, K: An Object Oriented Word Expert Parser for Japanese Language, Master Thesis, Dept. of Information Science, Tokyo Institute of Technology, in preparation.

[MY84] Matsumoto, Y.: Management of Industrial Software Production, Computer Vol. 17, No. 2, (1984), pp.59-72.

[OY84] Ohsawa, I. Yonezawa, A.: An Object Oriented Dialog Understanding System, Preprint of WGNL, IPSJ, July (1984), in Japanese.

[SP81] Special Issue For Distributed Problem Solving, IEEE Tans. on Systems, Man and Cybernetics, Vol. SMC-11, No.1, (1981).

[ST83] Shapiro, E. and Takeuchi, A.: Object Oriented Programming in Concurrent Prolog, New Generation Computing, Vol. 1, No. 1, (1983).

[TI84] Tokoro, M. and Ishikawa, Y.: An Object-Oriented Approach to Knowledge Systems, Proc. Int. Conf. on Fifth Generation Computer Systems, (1984).

[WM81] Weinreb, D. and Moon, D.: Flavors: Message Passing in the Lisp Machine, AI-Memo 602, MIT AI Lab., (1981).

[YM84] Yonezawa, A: An Actor Based Core Language ABCL and Its Example Programs, Research Report, Dept. of Information Science, Tokyo Institute of Technology, in preparation.

[YO79] Yonezawa, A.: Comments on Monitors and Path-Expressions, J. of Information processing, Vol. 1, No. 4, (1979), pp.180-186.

[YO84] Yonezawa, A.: On Object Oriented Programming, Computer Software, Vol. 1, No. 1, (1984), pp.29-41. in Japanese.

[YO84a] Yonezawa, A.: Parallel Discrete Event Simulation using a Concurrent Object Oriented Language, in preparation.

Appendix I Semantics of Now Type Message Passings

```
;; The semantics of now type message passing can be described in terms
;; of past type message passings and an appropriate continuation.


[object
    ...

   (=> [...]

   ... [T <== [<request>]] ...          ;a now type message passing,
                                        ;thus the C-part is nil.
       <rest of computation>)

  ...   ]
```

This is equivalent to

```
[object
    ...

   (=> [...]    ...

    [T <= [<request>
           .
          [object            ;the definition of the continuation begins here.
            (=> [Any]       ;any binds the result to be returned from T.
              ... Any ...           ;this occurrence of Any corresponds to
                                    ;the occurrence of [T <== [<request>]]
             <rest of computation>)]]])

   ...   ]
```

Appendix II Semantics of Future Type Message Passing

```
;; Semantics of future type message passing can be described in terms of
;; now type and past type message passings.


[object
    (state:  ...
        [x := <exp>]          ;suppose a future variable x is
            ...     )         ;declared and initialized.
  ...
  (=> [...]
    ...
        [x := [T <= [<request>]]]       ;a future type message passing.
                                        ;assuming the C-part is nil.
        ...  x  ...              )      ;x is referenced to here.
  ...                                 ]
```

This is semantically equivalent to the following:

```
  [object
      (state: ...
          [x = [CreateFutureVariable <== [init: <exp>]]]
              ...   )
      ...
      (=> [...]
         ...
          [x <= [to_be_updated:]]
          [T <= [<request> . x]]

          ... [x <== [contents?]] ...)     ; this now type message passing
      ...                               ]  ; corresponds to the reference to x
```

where

```
[object CreateFutureVariable
  (=> [init: Val]

    ![object
        (state: [contents := Val]
                [updated := t]
                [continuation := nil])

       (=> [to_be_updated:]
         [updated := nil]
         [continuation := nil])

       (=> [contents? . C]
         (case updated
          (is t
            [C <= [contents]])
          (otherwise
            [continuation := C])))

       (=> [New_value]
         [updated := t]
         [contents := New_value]
         (case (not_null continuation)
           (is t
             [continuation <= [contents]])))]
)]
```